Randomized Signature Sort: Implementation & Performance Analysis

Tamana Pathak
Department of computer Science and Engineering,
Thapar University, Patiala,
Punjab, India

Dr. Deepak Garg
Department of computer Science and Engineering,
Thapar University, Patiala,
Punjab, India

ABSTRACT
Recently the lower bound for integer sorting has considerably improved and achieved with comparison sorting $O(n \log n)$ to $O\left(n\sqrt{\log\log n}\right)$ [1] for a deterministic algorithms or to $O(n)$ for a radix sort algorithm in space that depends only on the number of input integers. Andersson et al. [2] presented signature sort in the expected linear time and space which gives very bad performance than randomized quick sort. We earlier presented in [14] that performance of signature sort can be enhanced using hashing and bitwise operators. This paper gives the implementation of that idea and later we have compared the performance of algorithm with existing randomized signature sort and randomized quick Sort.

General Terms
Algorithms, Complexity.

Keywords
Randomized algorithms; Integer Sorting; Linear Complexity.

1. INTRODUCTION
The lower bound $O\ (n\ log\ log\ n)$ given by Andersson et al. uses $O(m)$ extra memory. Later on, Yijie Han also improved the lower bound to $O\ (n\ log\ log\ n)$ with linear space. And he then introduced to a new lower bound of $O(n\sqrt{log\ log\ n})$ with linear space.

Using randomization the lower bound for deterministic algorithms can be optimized further as Andersson et al. presented a concept which takes $O(n\ log\ log\ n)$ and linear space. Later on, he came up with signature sort with expected linear time and space. But the relative performance of signature sort with traditional deterministic sorting algorithms is very poor.

The Randomized Signature Sorting algorithm works in two phases: One of the phases is Word Formation phase, packs multiple integers into a single machine word to sort quickly by operating on multiple integers with a single instruction. Another Phase is Comparison Sorting i.e. Sorting between words and Sorting within words, discussed later. To accomplish linear time sorting of n integers requires to pack $(log\ n\ log\ log\ n)$ of integers into one machine word.

The existing signature sort divided integers into fields and then packs them into different words to perform packed sorting. This concept consumes lot of running time of computer as integers must be divided into fields and then each field must be packed into different word. After that comparison is performed and based upon that the sorted list is generated. As we can see this signature sort requires lot of extra operation.

In this we have discussed the concept as well as the results from the implementation of improved randomized signature sort which reduces the extra operation required by the signature sort. Instead of dividing integer into fields we hash each integer into signature which is only $O(log\ n)$ bit size long. It reduces the requirement of dividing integers into field as integer has only one signature of reduced bit size. We can pack multiple signatures into one word. In this way, we only use single word for multiple integers, instead of using one word for one field of integer. Thus, the operation required after this in only comparison.

The Signature Sorting algorithm uses the signatures which are computed by applying hash function on the integer i.e. input values to reduce the size of the integers being sorted.

2. SIGNATURES
The purpose for the creation of signature is to decrease the size of integers to be operated upon; signatures have the lesser bit size than the original inputs. The signatures are computed for each input with a universal hash function. Such signatures must have the size of *O(log n)* where n is the number of input integers. These signatures are created by applying hash function on the input integers. The signatures created with this method must follow the property:

If $A_i \leq A_j\ \forall\ i,j\ \in\ \{0,1\ldots n-1\}$ then

$S_i \leq S_j\ \forall\ i,j\ \in\ \{0,1\ldots n-1\}$.

Where A_i's are the input integers, S_i's the corresponding signatures and n is the number of input integers [14].

3. HASHING
The hash function is applied on integers to reduce their size by creating signatures of $O(log\ n)$ bit size. The hash function must provide collision free hashing to ensure accurate and better result. The hash function must take $O(n)$ expected time. This will improve the overall performance of algorithm. We are implementing the following has function from [13] that is used for hashing of integers into signatures. Division must be avoided in hash function for better performance. The hash function is,

$$h_a(x) = (ax\ mod\ 2^k)/2^{k-l}$$

Where, k is the number of bits in the input integer, l is the number of bits in the signature which will

be$O(log\ n)$Error! Bookmark not defined. . a is randomly chosen between zero and 2^k.

Since the division in the above function is division by a power of two, it can be implemented as a left shift. This function will take $O(n)$ time. The above said hash function assures $O(log\ n)$ bit size of signatures and also collision free result [13].

4. WORD FORMATION

In Andersson's concept integers are divided into fields and each of these fields is packed into words. This is a bit of overhead as each field of integers is required to be packed and compared. We are using improved concept of word formation using hashing [14]. As discussed above hash function will hash whole integer into a signature with reduced bit size of $O(log\ n)$. After that the packing of multiple signatures into one word will be done.

It is an important phase as multiple integers i.e. signatures (hashes) of integers will be packed in a word. We must ensure that this phase runs error free while implementing.

If *w* is word size of the machine, *sb* is the number of bits in the signature, *wl* is the word length, *m* is the number of words, *l* is the number of signatures in a word and *n* is the number of input integers then:

$$l = wl/sb$$

$$m = ciel(n/l)$$

There will $O\ (log\ n\ log\ log\ n)$ number of words be created overall [14]. The word formation phase will take $O(n)$ time [14]. The following pseudo code is for word formation:

Word[] and Sign[] are the array of words and array of signatures respectively.

Step 1: Repeat for i=1 to m by 1.
Step 2: Repeat for j=1 to L by 1
Step 3: word[i] = (word[i])<<sb) | sign[*i* * *l* + j]
Step 4: Return.

5. COMPARISON SORTING

In this phase, actual sorting will be implemented. Here sorting means exchanging the positions of the signatures from one word to another word and also exchanging positions within word itself. The whole sorting procedure is divided into two sub-phases.

The first sub-phase is sorting between words. The sorting between words includes applying XOR and AND in such a way that the result will show which signatures need to be swapped. First, we will apply XOR on words and then apply XOR and AND operators on result with the word in which we want smallest of this signatures [14]. This operation will give non-zero value on corresponding bits where signatures are needed to be swapped. Later we will swap those signatures again using XOR operator. The overall expected time for this sub-phase is only $O(n)$. This is pseudo code for the first sub-phase:This function will take two words as input and sort them. Let the two words be x and y.

Sort(x,y)

Step1: Set temp= x XOR y.
Step2: Set temp= temp XOR x.
Step3: Set temp= temp AND y.
Step4: For all non-zero signatures in temp, swap the corresponding signatures in x and y.
Step 5: Return.

After first sub-phase the words will be in sorted order with each other, but there are multiple signatures in every word, hence we need to perform sorting within word itself. We will perform this comparison by recursively dividing a word into 2 halves until reaches to single signature in a half [14]. The expected time for this sub-phase will be *O (log m)* [14].

The algorithm for comparison within word is given as follows:

Step 1: Repeat for i=1 to log l.
Step 2: Divide word into 2^i parts.
Step 3: call sort () for each adjacent parts i.e. 1st, 2nd, 3rd, 4th and so on.
Step 4: Return.

6. UNPACKING

The unpacking implies when the sorting has been done there is a need to get back the original input integers from the sorted signatures. In order to perform this task there are two ways: one possible way is to keep track using index and another is to create reverse hash function. During the implementation of improved randomized signature sort we have promoted the usage of indexing. The indexes are swapped as the signatures are swapped therefore it's easier to sort the signatures and get back the original input.

Unpacking also calls for new techniques like reverse hash function. Reverse hash function can make it faster as in order to retain the sorted list, there is no need to maintain ranks or indexing would be required. This task might be tedious but very useful.

7. PERFORMANCE ANALYSIS

In this section we will compare the performance of Improved randomized Signature Sort with the Existing Signature Sort as well as Quick Sort. The comparison includes the CPU running time and the memory requirement. We are using 32-bit integers as input and 64-bit long data type provided by Java to store words. The input integers are generated using random function and stored in a file. The implementation of algorithms is done in Java6.0 on Eclipse-IDE using OOP approach. The platform used is Intel 64-bit with Core i3 processor having a frequency of 2.40 GHz with Windows 7(64-bit) Enterprise Edition running on it. The System had a RAM of 3GB. While measuring the

performance i.e. collecting the details all other processes were terminated.

Runtime Comparison: The runtime of the algorithm is measured as CPU milliseconds using currentTimeMillis() which is provided by the System Class. The running time includes reading input from a file and writing output to output file. The runtime also includes hashing of integers into signatures, packing of signatures into words and extracting signatures from words. The table 1 shows the data collected of running time of algorithms.

Table 1. CPU Runtime Comparison

N	Existing RSS	Quick sort	Improved RSS
50	1	1	1
100	2	4	5
200	4	15	11
300	12	16	16
400	18	17	22
500	24	18	31
600	32	20	32
700	40	22	32
800	54	25	44
900	59	29	46
1000	76	31	47
2000	206	52	94
4000	623	83	234

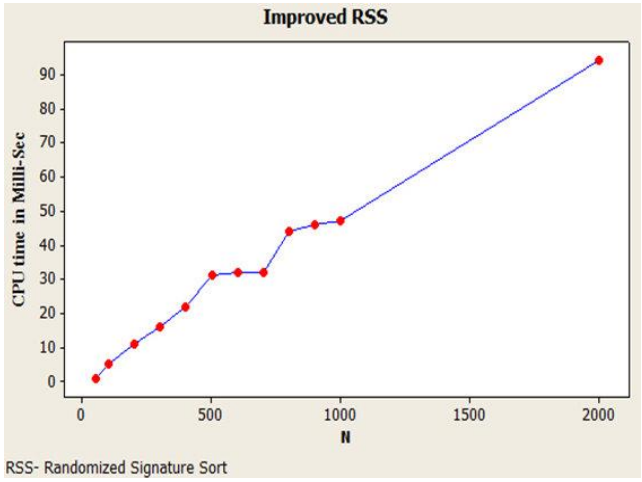

Figure 1 Runtime plot of Improved Randomized Signature Sort

The above figure 1 shows that the graph when plotted between CPU time in Milliseconds and number of input integers grows close to linear. As the input size increases the CPU time also increases proportionally.

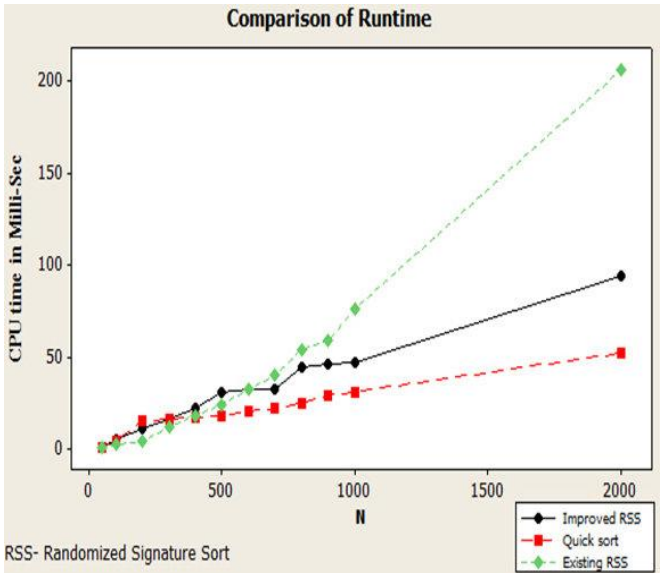

Figure 2: Comparison of Runtime

Whereas if we look at above figure 2 we can see the comparative performance with respect to running time of existing Randomized signature Sort, Quick Sort and Improved Randomized Signature sort

Memory Comparison: We ran the randomized Signature Sort algorithms on a variety of input sequences to compare its performance of memory requirement to Quick sort. We measured the memory consumed for different inputs by looking the values corresponding to the process 'javaw.exe' as the algorithms is run in Java language.

The table 2 listed the data collected of memory required for both Quick sort as well as Improved Randomized Signature Sort:

Table 2. Memory Comparison

N	Quick sort	Improved RSS
50	4192	4280k
100	4252	4300k
200	4768	4796k
300	4864	4808k
400	4884	4848k
500	4908k	4948k
600	4912k	4964k
700	4940k	4988k
800	4968k	5012k
900	5024k	5036k
1000	5052k	5060k
2000	5244k	5284k
4000	5644k	5692k

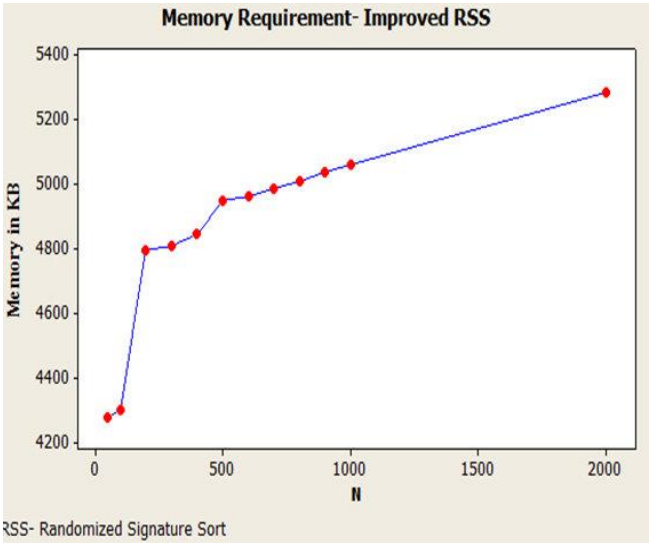

Figure 3: Memory Requirement of MRSS

The above graph depicts that as the input sequence lengths increases the amount of memory required is also increases considerably. As the input size grows, the graph for improved randomized signature sort appears to take nearly linear memory size.

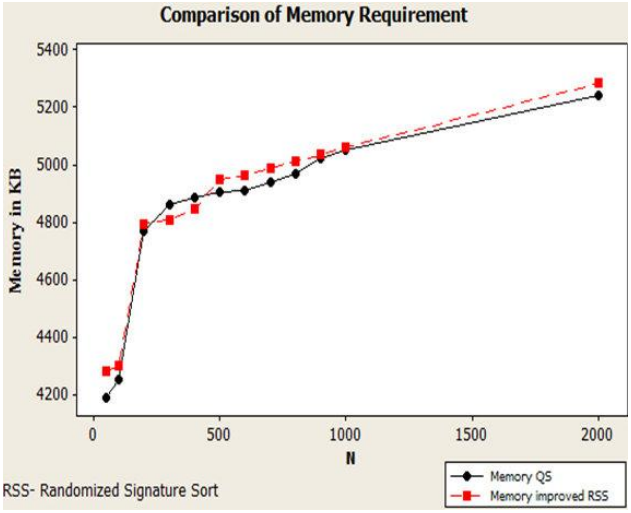

Figure 4: Comparison of Memory Requirement

The above figure 4 shows the performance of both improved randomized signature sort and the Quick sort. As the input size grows, the graph for both algorithms also grows close to each other. The difference between the two is minimal thus we can say that existing randomized signature sort has been improved considerably depleting the large difference of performance compared to Quick Sort.

8. CONCLUSION

We finally come up with implementation of improved randomized signature sort, which is not only stable but better in performance. With new algorithm to perform randomized signature sort, number of comparisons also have been reduced as there is no need to consider each integer for comparison for lesser number of time than Existing randomized signature sort. As in the existing one, integers were divided into fields and each field had to be considered for comparison operation, which overall acted like an overhead on the algorithm. Hence by reducing the number of comparison to be performed, algorithm for improved randomized signature sort runs more efficiently.

The implementation of algorithm gives $O(n)$ expected time which uses only linear space. The actual running time of this variant is comparatively very low than existing signature sort. But still slightly less efficient than randomized quick sort. The use of bitwise operators and hashing has improved the performance of sorting algorithm significantly. There is a scope to improve it to make its performance more close to traditional Quick Sort.

9. REFERENCES

[1] Yijie Han and Mikkel Thorup. Integer sorting in $O(n\sqrt{log\ log\ n})$ expected time and linear space. In *IEEE Symp. on Foundations of Computer Science*, volume 43, 2002.

[2] Andersson, Hagerup, Nilsson, and Raman. Sorting in linear time? In *STOC: ACM Symposium on Theory of Computing (STOC)*, 1995.

[3] D. Kirkpatrick and S. Reisch. Upper bounds for sorting integers on random access machines. 1984.

[4] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees.1993. Announced at STOC'90.

[5] W. Paul and J. Simon. Decision trees and random access machines. In Proc. Symp. ¨uber Logik and Algoritmik, 1980.

[6] L. J. Comrie. The hollerith and powers tabulating machines.Trans. Office Machinary Users' Assoc., Ltd, 1929-30.

[7] A. I. Dumey. Indexing for rapid random access memory systems. Computers and Automation, 1956.

[8] Mikkel Thorup. Randomized sorting in *O (n log log n)* time and linear space using addition, shift, & bit-wise boolean operations.

[9] Y. Han: Deterministic Sorting in *O (n log log n)* Time and Linear Space, J. Algorithms 50(1): 2004.

[10] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein: Introduction to Algorithms, Second Edition, The MIT Press and McGraw-Hill Book Company2001.

[11] S. Cook and R. Reckhow. Time-bounded random access machines. J. Comp. Syst. Sc., 10(2):1973.

[12] M. Dietzfelbinger, T. Hagerup, J. Katajainen, M. Penttonen, A reliable randomized algorithm for the closest-pair problem, J. Algorithms 25 (1997).

[13] B. Vandiver, A.Rolfe, Exploiting sleight-of model to achieve super-luminal sorting: 2003

[14] T. Pathak, D. Garg, Improving performance of Randomized Signature Sort using hashing and Bitwise operators, JGRCS Vol 2 No 3, 2011.